第 3 章 VHDL 程序结构

如何才算一个完整的 VHDL 程序，或者说设计实体，并没有完全一致的结论，因为不同的程序设计目的可以有不同的程序结构。例如，对于在综合后具有相同逻辑功能的 VHDL 程序，设计者注重于系统的行为仿真和仅注重综合后的时序仿真，对程序结构的要求是不一样的，因为后者无需在程序中加入控制仿真的语句及设置相关的参数。

通常可以认为，一个完整的设计实体的最低要求应该能为 VHDL 综合器所接受，并能作为一个独立设计单元，即元件的形式而存在的 VHDL 程序。这里的所谓元件，既可以被高层次的系统所调用，成为该系统的一部分，也可以作为一个电路功能块而独立存在和独立运行。

图 2-5 只是一般意义上的 VHDL 结构模式，并不是必须具备的模式。在 VHDL 程序中，实体（ENTITY）和结构体（ARCHITECTURE）这两个基本结构是必需的，它们可以构成最简单的 VHDL 程序。实体是设计实体的组成部分，它包含了对设计实体输入和输出的定义和说明，而设计实体则包含了实体和结构体两个在 VHDL 程序中的最基本的部分。通常，最简单的 VHDL 程序结构中还应包括另一重要的部分，即库（LIBRARY）和程序包（PACKAGE）。一个实用的 VHDL 程序可以由一个或多个设计实体构成，可以将一个设计实体作为一个完整的系统直接利用，也可以将其作为其它设计实体的一个低层次的结构，即元件来例化（元件调用和连接），就是用实体来说明一个具体的器件。图 2-5 中配置（CONFIGURATION）结构的设置，常用于行为仿真中，如用于对特定结构体的选择控制。VHDL 程序结构的一个显著特点就是，任何一个完整的设计实体都可以分成内外两个部分，外面的部分称为可视部分，它由实体名和端口组成；里面的部分称为不可视部分，由实际的功能描述组成。一旦对已完成的设计实体定义了它的可视界面后，其它的设计实体就可以将其作为已开发好的成果直接调用，这正是一种基于自顶向下的多层次的系统设计概念的实现途径。

§3.1 实 体（ENTITY）

实体作为一个设计实体的组成部分，其功能是对这个设计实体与外部电路进行接口描述。实体是设计实体的表层设计单元，实体说明部分规定了设计单元的输入输出接口信号或引脚，它是设计实体对外的一个通信界面。就一个设计实体而言，外界所看到的仅仅是它的界面上的各种接口。设计实体可以拥有一个或多个结构体，用于描述此设计实体的逻辑结构和逻辑功能。对于外界来说，这一部分是不可见的。

不同逻辑功能的设计实体可以拥有相同的实体描述，这是因为实体类似于原理图中的一个部件符号，而其具体的逻辑功能是由设计实体中结构体的描述确定的。实体是 VHDL 的基本设计单元，它可以对一个门电路、一个芯片、一块电路板乃至整个系统进行接口描述。

1. 实体语句结构

以下是实体说明单元的常用语句结构：

```
ENTITY  实体名  IS
   [GENERIC ( 类属表 ); ]
   [PORT ( 端口表 ); ]
END  ENTITY 实体名;
```

实体说明单元必须按照这一结构来编写，实体应以语句“ENTITY 实体名 IS”开始，以语句“END ENTITY 实体名 ;”结束，其中的实体名可以由设计者自己添加。中间在方括号内的语句描述，在特定的情况下并非是必需的。例如构建一个 VHDL 仿真测试基准等情况中可以省去方括号中的语句。对于 VHDL 的编译器和综合器来说，程序文字的大小写是不加区分的，但为了便于阅读和分辨，建议将 VHDL 的标识符或基本语句关键词以大写方式表示，而由设计者添加的内容可以以小写方式来表示，如实体的结尾可写为“END ENTITY nand”，其中的 nand 即为设计者取的实体名。

2. 实体名

一个设计实体无论多大和多复杂，在实体中定义的实体名即为这个设计实体的名称。在例化（已有元件的调用和连接）中，即可以用此名对相应的设计实体进行调用。例 2-3 中的 1 位全加器的设计就是在其结构体中调用了描述或门和半加器的设计实体，在其例化操作中，直接使用了它们的实体名（注意，不是该文件的文件名）。例如：

【程序 3-1】

```
...
COMPONENT h_adder                    -- 元件调用说明
    PORT (  a，b : IN STD_LOGIC ;
          co，so : OUT STD_LOGIC );
END COMPONENT;
...
u1 :  h_adder PORT MAP ( a =>ain，b =>bin，co=>d，so =>e);
...                                  -- 这里的符号“=>”是端口关联符号
```

此例中调用的元件名 h_adder 即为程序 2-3 中半加器的实体名。

有的 EDA 软件对 VHDL 文件的取名有特殊要求，如 MAX+PLUSII 要求文件名必须与实体名一致，如 h_adder.vhd。一般地，将 VHDL 程序的文件名取为此程序的实体名是一种比较好的编程习惯。

3. GENERIC 类属说明语句

类属（GENERIC）参量是一种端口界面常数，常以一种说明的形式放在实体或块结构体前的说明部分。类属为所说明的环境提供了一种静态信息通道。类属与常数不同，常数只能从设计实体的内部得到赋值，且不能再改变，而类属的值可以由设计实体外部提供。因此，设计者可以从外面通过类属参量的重新设定而容易地改变一个设计实体或一个元件的内部电路结构和规模。

类属说明的一般书写格式如下：

```
GENERIC([ 常数名 : 数据类型 [ : 设定值 ]
 { ; 常数名 : 数据类型 [ : 设定值 ] } ) ;
```

类属参量以关键词 GENERIC 引导一个类属参量表，在表中提供时间参数或总线宽度等静态信息。类属表说明用于设计实体和其外部环境通信的参数，传递静态的信息。类属在所定义的环境中的地位与常数十分接近，但却能从环境（如设计实体）外部动态地接受赋值，其行为又有点类似于端口 PORT。因此常如以上的实体定义语句那样，将类属说明放在其中，且放在端口说明语句的前面。

在一个实体中定义的、来自外部赋入类属的值可以在实体内部或与之相应的结构体中读到。对于同一个设计实体，可以通过 GENERIC 参数类属的说明，为它创建多个行为不同的逻辑结构。比较常见的情况是利用类属来动态规定一个实体的端口的大小，或设计实体的物理特性，或结构体中的总线宽度，或设计实体中底层中同种元件的例化数量等等。

一般在结构体中，类属的应用与常数是一样的。例如，当用实体例化一个设计实体的器件时，可以用类属表中的参数项定制这个器件，如可以将一个实体的传输延迟、上升和下降延时等参数加到类属参数表中，然后根据这些参数进行定制，这对于系统仿真控制是十分方便的。其中的常数名是由设计者确定的类属常数名，数据类型通常取 INTEGER 或 TIME 等类型，设定值即为常数名所代表的数值。但需注意，VHDL 综合器仅支持数据类型为整数的类属值。

程序 3-2 和 3-3 是两个使用了类属说明的实例描述。

【程序 3-2】

```
ENTITY  mcu1  IS
   GENERIC (addrwidth : INTEGER := 16);
   PORT(
     add_bus : OUT STD_LOGIC_VECTOR(addrwidth-1 DOWNTO 0) );
     ...
```

在这里，GENERIC 语句对实体 mcu1 作为地址总线的端口 add_bus 的数据类型和宽度作了定义，即定义 add_bus 为一个 16 位的标准位矢量，定义 addrwidth 的数据类型是整数 INTEGER 。其中，常数名 addrwidth 减 1 即为 15，所以这类似于将上例端口表写成：

```
PORT (add_bus  :  OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
```

由程序 3-2 可见，对于类属值 addrwidth 的改变将对结构体中所有相关的总线的定义同时作了改变，由此将改变整个设计实体的硬件结构。

【程序 3-3】 2 输入与门的实体描述。

```
ENTITY  PGAND2  IS
   GENERIC (    trise : TIME := 1 ns;
                tfall : TIME := 1 ns  ) ;
      PORT (    a1 :  IN STD_LOGIC ;
                a0 :  IN STD_LOGIC ;
                z0 :  OUT STD_LOGIC  );
END ENTITY PGAND2;
```

这是一个准备作为 2 输入与门的设计实体的实体描述，在类属说明中定义参数 trise 为上沿宽度；tfall 为下沿宽度，它们分别为 1ns，这两个参数用于仿真模块的设计。

以下的程序 3-5 是一个顶层设计文件，它在例化语句中调用了程序 3-4。读者应注意到，在程序 3-4 中的类属变量 n 并没有如程序 3-2 那样明确规定了它的取值，n 的具体取值是在程序 3-5 中的类属映射语句 GENERIC MAP ( )中指定的，并在两个不同的类属映射语句中作了不同的赋值。

程序 3-4 和 3-5 给出了类属语句的一种典型应用。显然，类属语句的应用，为方便而迅速地改变电路的结构和规模提供了极便利的条件。

【程序 3-4】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY andn IS
   GENERIC (  n : INTEGER );
        PORT(a : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             c : OUT STD_LOGIC);
 END;
 ARCHITECTURE behav OF andn IS
   BEGIN
     PROCESS (a)
         VARIABLE int : STD_LOGIC;
      BEGIN
         int := '1';
         FOR i IN a'LENGTH - 1 DOWNTO 0 LOOP
            IF a(i)='0' THEN
                 int := '0';
            END IF;
         END LOOP;
           c <=int ;
       END PROCESS;
  END;
```

【程序 3-5】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY exn IS
   PORT(d1,d2,d3,d4,d5,d6,d7 : IN STD_LOGIC;
                        q1,q2  : OUT STD_LOGIC);
END;
 ARCHITECTURE exn_behav OF exn IS
    COMPONENT andn
      GENERIC (  n : INTEGER);
        PORT(a: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
             c: OUT STD_LOGIC);
     END COMPONENT ;
 BEGIN
   u1: andn GENERIC MAP (n =>2)
    PORT MAP (a(0)=>d1,a(1)=>d2,c=>q1);
```

```
    u2: andn GENERIC MAP (n =>5)
          PORT MAP (a(0)=>d3,a(1)=>d4,a(2)=>d5,
                    a(3)=>d6,a(4)=>d7, c=>q2);
    END;
```

程序 3-5 给出了类属映射语句 GENERIC MAP ( )配合端口映射语句 PORT MAP ( ) 语句的使用范例。端口映射语句是本结构体对外部元件调用和连接过程中，描述元件间端口的衔接方式的，而类属映射语句具有相似的功能，它描述相应元件类属参数间的衔接和传送方式，读者不妨利用程序 3-5 中 GENERIC MAP ( )的使用方法，作一些相关的练习。

4. PORT 端口说明

由 PORT 引导的端口说明语句是对一个设计实体界面的说明。其端口表部分对设计实体与外部电路的接口通道进行了说明，其中包括对每一接口的输入输出模式（MODE，或称端口模式）和数据类型（TYPE）进行了定义。在实体说明的前面，可以有库的说明，即由关键词“LIBRARY”和“USE”引导一些对库和程序包使用的说明语句，其中的一些内容可以为实体端口数据类型的定义所用。

实体端口说明的一般书写格式如下：

```
PORT ( 端口名 : 端口模式  数据类型 ;
     { 端口名 : 端口模式  数据类型} ) ;
```

其中的端口名是设计者为实体的每一个对外通道所取的名字，端口模式是指这些通道上的数据流动方式，如输入或输出等。数据类型是指端口上流动的数据的表达格式或取值类型，这是由于 VHDL 是一种强类型语言，即对语句中的所有的端口信号、内部信号和操作数的数据类型有严格的规定，只有相同数据类型的端口信号和操作数才能相互作用。

一个实体通常有一个或多个端口，端口类似于原理图部件符号上的管脚。实体与外界交流的信息必须通过端口通道流入或流出。程序 3-6 是一个 2 输入与非门的实体描述示例，图 3-1 是它对应的原理图。

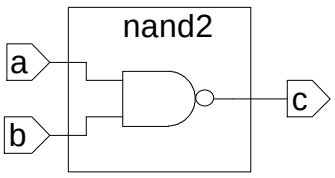


图 3-1 nand 对应的原理图符号

【程序 3-6】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY nand2 IS
    PORT(a : IN  STD_LOGIC ;
         b : IN  STD_LOGIC ;
         c : OUT STD_LOGIC ) ;
END nand2 ;
...
```

图 3-1 中的 nand2 可以看成一个设计实体，它的外部接口界面由输入输出信号端口 a、b 和 c 构成，内部逻辑功能是一个与非门。在电路图上，端口对应于器件符号的外部引脚。端口名作为外部引脚的名称，端口模式用来定义外部引脚的信号流向。IEEE 1076 标准程序包中定义了以下的常用端口模式：

• IN 模式：IN 定义的通道确定为输入端口，并规定为单向只读模式，可以通过此端口将变量（Variable）信息或信号（Signal）信息读入设计实体中。

• OUT 模式：OUT 定义的通道确定为输出端口，并规定为单向输出模式，可以通过此端口将信号输出设计实体，或者说可以将设计实体中的信号向此端口赋值。

• INOUT 模式：INOUT 定义的通道确定为输入输出双向端口，即从端口的内部看，可以对此端口进行赋值，也可以通过此端口读入外部的数据信息；而从端口的外部看，信号既可以从此端口流出，也可以向此端口输入信号。INOUT 模式包含了 IN、OUT 和 BUFFER 三种模式，因此可替代其中任何一种模式，但为了明确程序中各端口的实际任务，一般不作这种替代。

程序 3-7 中将 MCS51 单片机的数据口 P0 的工作模式定义为 INOUT，因此在程序中很方便地实现了 P0 口作为可读可写的双向端口的功能。

• BUFFER 模式：BUFFER 定义的通道确定为具有数据读入功能的输出端口，它与双

【程序 3-7】

```
...
ENTITY MCS51 IS
  PORT (                                        -- 与 8031 接口的各端口定义 :
    P0 : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);  -- 双向地址/数据口
    P2 : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);   -- 高 8 位地址线
    RD，WR : IN STD_LOGIC;                        -- 读、写允许
...
END MCS51;
...
   PROCESS( WR_ENABLE2 )
     BEGIN
       IF WR_ENABLE2'EVENT AND WR_ENABLE2 = '1'
         THEN LATCH_OUT2 <= P0;  END IF;  --从 P0 口读入外部信息
   END PROCESS;
  PROCESS( P2,LATCH_ADDRES,READY,RD )
     BEGIN
      IF (LATCH_ADDRES="01111110") AND (P2="10011111")
             AND (READY='1') AND (RD='0')  THEN
             P0 <= LATCH_IN1 ; -- 寄存器中的数据输入 P0 口，由 P0 向外输出
        ELSE P0 <= "ZZZZZZZZ" ;
     END IF;  -- 禁止读数，P0 口输出呈高阻态
   END PROCESS;
...
```

向端口的区别在于只能接受一个驱动源。BUFFER 模式从本质上将仍是 OUT 模式，只是在内部结构中具有将输出至外端口的信号回读的功能，即允许内部回读输出的信号，即允许反馈。如计数器的设计，可将计数器输出的计数信号回读，以作下一计数值的初值。与 INOUT 模式相比，显然，BUFFER 的区别在于回读（输入）的信号不是由外部输入的，而是由内部产生，向外输出的信号，有时往往在时序上有所差异。

通常实现内部反馈有两种方式，即利用 BUFFER 建立一个缓冲模式的端口，如程序 3-8 所示；或在结构体内定义一个缓冲节点信号 SIGNAL，如程序 3-9 所示。它们的逻辑功

能和综合后的电路都是一样的，图 3-2 就是程序 3-8 或 3-9 综合后的逻辑电路。由图 3-2 可以看出，BUFFER 并不总是一个特定的端口，而可能是一种电路的接口方式。

【程序 3-8】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY bfexp IS
   PORT(
         clk,rst,din : IN  STD_LOGIC ;
                  q1 : BUFFER  STD_LOGIC ;
                  q2 : OUT STD_LOGIC
         ) ;
END bfexp ;
ARCHITECTURE behav1 OF bfexp IS
   BEGIN
     PROCESS(clk,rst)
       BEGIN
         IF rst ='0'  THEN
           q1 <= '0' ;
           q2 <= '0' ;
        ELSIF  clk'EVENT AND clk = '1' THEN
           q1 <= din ;  --将由 din 读入的数据向 q1 输出
           q2 <= q1 ;   --将向 q1 输出的数据回读，并向 q2 赋值
         END IF;
      END PROCESS;
    END;
```

【程序 3-9】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY bfexp IS
   PORT(clk,rst,din : IN  STD_LOGIC ;
                  q1 : OUT STD_LOGIC ;
                  q2 : OUT STD_LOGIC ) ;
END bfexp ;
ARCHITECTURE behav1 OF bfexp IS
   SIGNAL qbuf : STD_LOGIC;  --定义数据暂存缓冲信号 qbuf
   BEGIN
     PROCESS(clk,rst)
       BEGIN
         IF rst ='0'  THEN
           qbuf <= '0' ;
             q2 <= '0' ;
        ELSIF  clk'EVENT AND clk = '1'   THEN
           qbuf <= din ;  --将由 din 读入的数据暂存于 qbuf
           q2 <= qbuf ;   --将缓冲信号 qbuf 中的数据向 q2 赋值
         END IF;
       q1 <= qbuf;       --将缓冲信号 qbuf 中的数据向 q1 赋值，并由此输出
    END PROCESS;
    END;
```

综上所述，在实际的数字集成电路中，IN 相当于只可输入的引脚，OUT 相当于只可输出的引脚，BUFFER 相当于带输出缓冲器并可以回读的引脚（与 TRI 引脚不同），而 INOUT

相当于双向引脚（即 BIDIR 引脚），是普通输出端口（OUT）加入三态输出缓冲器和输入缓冲器构成的。表 3-1 列出了端口的功能。

表 3-1 端口模式说明

端口模式	端口模式说明（以设计实体为主体）
IN	输入，只读模式
OUT	输出，单向赋值模式
BUFFER	具有读功能的输出模式，（从内部看）可以读或写，只能有一个驱动源
INOUT	双向，（从内部或外部看都）可以读或写

在实用中，端口描述中的数据类型主要有两类：位（BIT）和位矢量（BIT_VECTOR）。若端口的数据类型定义为 BIT，则其信号值是一个 1 位的二进制数，取值只能是 0 或 1，如程序 3-3 的端口 a0、a1 和 z0 的数据类型是 STD_LOGIC，这是取自 IEEE 库中 STD_LOGIC_1164 程序包中 BIT 数据类型的定义；若端口的数据类型定义为 BIT_VECTOR，则其信号值是一组二进制数。如程序 3-2 的 add_bus 定义为一组 16 位的二进制数，从而构成一个地址总线端口。add_bus 的数据类型 STD_LOGIC_VECTOR 也是取自 IEEE 库中 STD_LOGIC_1164 程序包中标准位矢量数据类型的定义。

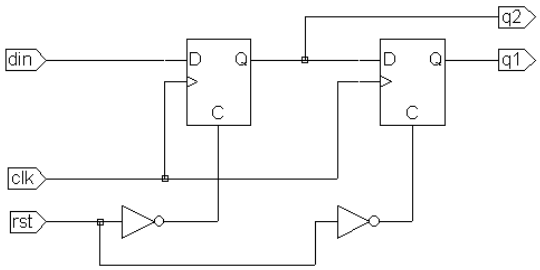


图 3-2 程序 3-8 或 3-9 综合后的电路图

§3.2 结构体（ARCHITECTURE）

由图 2-5 可知，结构体是实体所定义的设计实体中的一个组成部分。结构体描述设计实体的内部结构和/或外部设计实体端口间的逻辑关系。结构体由两大部分组成：

- 对数据类型、常数、信号、子程序和元件等元素的说明部分
- 描述实体逻辑行为的，以各种不同的描述风格表达的功能描述语句，它们包括各种形式的顺序描述语句和并行描述语句
- 以元件例化语句为特征的外部元件（设计实体）端口间的连接方式（如程序 2-3）

结构体将具体实现一个实体。每个实体可以有多个结构体，每个结构体对应着实体不同的结构和算法实现方案，其间的各个结构体的地位是同等的，它们完整地实现了实体的行为。但同一结构体不能为不同的实体所拥有。结构体不能单独存在，它必须有一个界面说明，即一个实体。对于具有多个结构体的实体，必须用 CONFIGURATION 配置语句指明用于综合的结构体和用于仿真的结构体。即在综合后的可映射于硬件电路的设计实体中，一个实体只能对应一个结构体。在电路中，如果实体代表一个器件符号，则结构体描述了这个符号的内部行为。当把这个符号例化成一个实际的器件安装到电路上时，则需配置语句为这个例化的器件指定一个结构体（即指定一种实现方案），或由编译器自动选一

个结构体。

1. 结构体的一般语言格式

结构体的语句格式如下：

```
ARCHITECTURE 结构体名 OF 实体名 IS
    [说明语句]
BEGIN
    [功能描述语句]
END ARCHITECTURE 结构体名;
```

在书写格式上，实体名必须是所在设计实体的名字，而结构体名可以由设计者自己选择，但当一个实体具有多个结构体时，结构体的取名不可相重。结构体的说明语句部分必须放在关键词“ARCHITECTURE”和“BEGIN”之间，结构体必须以“END ARCHITECTURE 结构体名;”作为结束句。

如程序 2-2 中的实体名是 Latch，结构体名是 one。在说明语句部分，定义了一个信号 sig_save ，它的数据类型定义为 STD_LOGIC。

结构体内部构造的描述层次和描述内容可以用图 3-3 来说明，它只是对结构体的内部构造作了一般的描述，并非所有的结构体必须同时具有如图 3-3 所示的所有的说明语句结构。一般地，如图 3-3 所示，一个完整的结构体由两个基本层次组成，即说明语句和功能描述语句两部分。

2. 结构体说明语句

结构体中的说明语句是对结构体的功能描述语句中将要用到的信号(SIGNAL)、数据类型(TYPE)、常数(CONSTANT)、元件(COMPONENT)、函数(FUNCTION)和过程(PROCEDURE)等加以说明。需要注意的是，在一个结构体中说明和定义的数据类型、常数、元件、函数和过程只能用于这个结构体中。如果希望这些定义也能用于其它的实体或结构体中，需要将其作为程序包来处理。

3. 功能描述语句结构

如图 3-3 所示的功能描述语句结构可以含有五种不同类型的以并行方式工作的语句结构。这可以看成是结构体的五个子结构。而在每一语句结构的内部可能含有并行运行的逻辑描述语句或顺序运行的逻辑描述语句。这就是说，这五种语句结构本身是并行语句，但它们内部所包含的语句并不一定是并行语句，如进程语句内所包含的是顺序语句。

图 3-3 中的五种语句结构的基本组成和功能分别是：

- 块语句是由一系列并行执行语句构成的组合体，它的功能是将结构体中的并行语句组成一个或多个子模块。
- 进程语句定义顺序语句模块，用以将从外部获得的信号值，或内部的运算数据向其它的信号进行赋值。
- 信号赋值语句将设计实体内的处理结果向定义的信号或界面端口进行赋值。

- 子程序调用语句用以调用过程或函数，并将获得的结果赋值于信号。
- 元件例化语句对其它的设计实体作元件调用说明，并将此元件的端口与其它的元件、信号或高层次实体的界面端口进行连接。

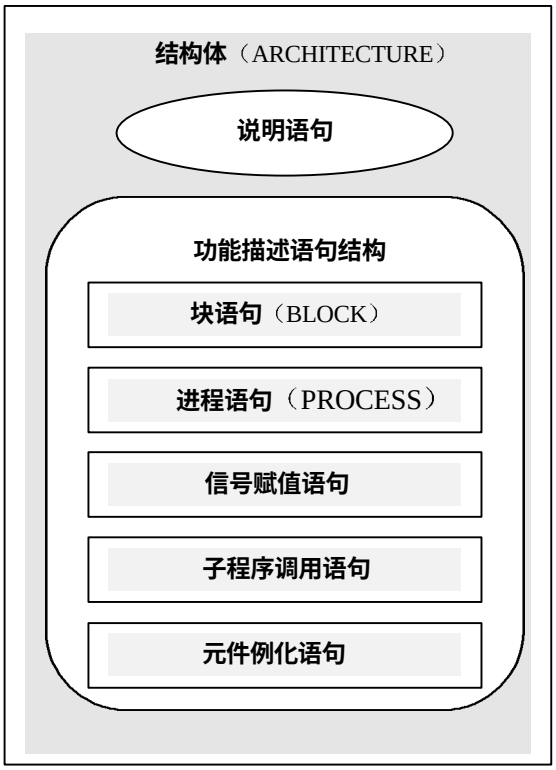


图 3-3 结构体构造图

下面的程序 3-10 是与程序 3-3 实体 PGAND2 对应的一个结构体，它的结构体名是 behav，结构体内有一个进程语句子结构，在此结构中用顺序语句描述了与门的输入信号 a0 和 a1 与输出信号 z0 间的逻辑关系，以及它们的时延关系。

【程序 3-10】

```
ARCHITECTURE behav OF PGAND2 IS
 BEGIN
   PROCESS (a1, a0)
    VARIABLE zdf : STD_LOGIC ;
     BEGIN
        zdf  :=  a1 AND a0 ;                  -- 向变量赋值
        IF  zdf ='1'  THEN
           z0 <= TRANSPORT zdf AFTER trise ;
        ELSIF  zdf ='0'  THEN
           z0 <= TRANSPORT zdf AFTER tfall ;
        ELSE
           z0 <= TRANSPORT zdf ;
     END IF ;
  END PROCESS ;
END ARCHITECTURE behav ;
```

请注意，VHDL 综合器将不支持或忽略此例中的时延关系，如“AFTER tfall”。

§3.3 块语句结构（BLOCK）

块（BLOCK）的应用类似于利用 PROTEL98 画一个大的电路原理图时，可以将一个总的原理图分成多个子模块，则这个总的原理图成为一个由多个子模块原理图连接而成的顶层模块图，而每一个子模块可以是一个具体的电路原理图。但是，如果子模块的原理图仍然太大，还可将它变成更低层次的原理图模块的连接图（BLOCK 嵌套）。显然，按照这种方式划分结构体仅是形式上的，而非功能上的改变。事实上，将结构体以模块方式划分的方法有多种，使用元件例化语句也是一种将结构体的并行描述分成多个层次的方法，其区别只是后者涉及到多个实体和结构体，且综合后硬件结构的逻辑层次有所增加。

实际上，结构体本身就等价于一个 BLOCK，或者说是一个功能块。BLOCK 是 VHDL 中具有的一种划分机制，这种机制允许设计者合理地将一个模块分为数个区域，在每个块都能对其局部信号、数据类型和常量加以描述和定义。任何能在结构体的说明部分进行说明的对象都能在 BLOCK 说明部分中进行说明

BLOCK 语句应用只是一种将结构体中的并行描述语句进行组合的方法，它的主要目的是改善并行语句及其结构的可读性，或是利用 BLOCK 的保护表达式关闭某些信号。

1. BLOCK 语句的格式

BLOCK 语句的表达格式如下：

```
块标号 : BLOCK [（块保护表达式）]
        接口说明
            类属说明
            BEGIN
```

并行语句
END BLOCK 块标号 ;

作为一个 BLOCK 语句结构，在关键词“BLOCK”的前面必须设置一个块标号，并在结尾语句“END BLOCK”右侧也写上此标号（此处的块标号不是必需的）。

接口说明部分有点类似于实体的定义部分，它可包含由关键词 PORT、GENERIC、PORT MAP 和 GENERIC MAP 引导的接口说明等语句，对 BLOCK 的接口设置以及与外界信号的连接状况加以说明。这类似于原理图间的图示接口说明。

块的类属说明部分和接口说明部分的适用范围仅限于当前 BLOCK。所以，所有这些在 BLOCK 内部的说明对于这个块的外部来说是完全不透明的，即不能适用于外部环境，或由外部环境所调用，但对于嵌套于更内层的块却是透明的，即可将信息向内部传递。块的说明部分可以定义的项目主要有：

- 定义 USE 语句
- 定义子程序
- 定义数据类型
- 定义子类型
- 定义常数
- 定义信号
- 定义元件

块中的并行语句部分可包含结构体中的任何并行语句结构。BLOCK 语句本身属并行语句，BLOCK 语句中所包含的语句也是并行语句。

2. BLOCK 的应用

BLOCK 的应用可使结构体层次鲜明，结构明确。利用 BLOCK 语句可以将结构体中的并行语句划分成多个并列方式的 BLOCK，每一个 BLOCK 都像一个独立的设计实体，具有自己的类属参数说明和界面端口，以及与外部环境的衔接描述。以下是两个使用 BLOCK 语句的实例，程序 3-11 给出了 BLOCK 语句的一个使用示例，而程序 3-12 描述了一个具有块嵌套方式的 BLOCK 语句结构。

在较大的 VHDL 程序的编程中，恰当的块语句的应用对于技术交流、程序移植、排错和仿真都是有益的。

【程序 3-11】

```
 ...
ENTITY gat IS
   GENERIC(l_time : TIME ;  s_time : TIME ) ; -- 类属说明
 PORT (b1, b2, b3 : INOUT BIT) ;              -- 结构体全局端口定义
 END ENTITY gat  ;
 ARCHITECTURE func OF gat IS
    SIGNAL  a1 : BIT ;                        -- 结构体全局信号 a1 定义
    BEGIN
Blk1 : BLOCK                                  -- 块定义，块标号名是 blk1
    GENERIC (gb1, gb2 : Time) ;               -- 定义块中的局部类属参量
    GENERIC MAP (gb1 => l_time,gb2 => s_time) ; -- 局部端口参量设定
```

```
    PORT (pb : IN BIT; pb2 : INOUT BIT );    -- 块结构中局部端口定义
    PORT MAP (pb1 => b1, pb2 => a1 ) ;       -- 块结构端口连接说明
    CONSTANT delay : Time := 1 ms ;          -- 局部常数定义
    SIGNAL s1 : BIT ;                        -- 局部信号定义
    BEGIN
    s1 <= pb1 AFTER delay ;
    pb2 <= s1 AFTER gb1, b1 AFTER gb2 ;
    END BLOCK blk1 ;
END ARCHITECTURE func ;
```

程序 3-11 只是对 BLOCK 语句结构的一个说明，其中的一些赋值实际上是不需要的。

【程序 3-12】

```
  ...
b1 :  BLOCK
    SIGNAL s1: BIT ;
    BEGIN
    S1 <= a AND b ;
b2 :  BLOCK
    SIGNAL s2: BIT ;
    BEGIN
      s2 <= c AND d ;
      b3 : BLOCK
       BEGIN
          Z <= s2 ;
       END BLOCK b3 ;
    END BLOCK b2 ;
      y <= s1 ;
  END BLOCK b1 ;
  …
```

程序 3-12 在不同层次的块中定义了同名的信号，显示了信号的有效范围。

3. BLOCK 语句在综合中的地位

与大部分的 VHDL 语句不同，BLOCK 语句的应用，包括其中的类属说明和端口定义，都不会影响对原结构体的逻辑功能的仿真结果。如以下的程序 3-13 和程序 3-14 的仿真结果是完全相同的。

【程序 3-13】

```
a1 : out1 <= '1' after 3 ns ;
blk1 : BLOCK
   BEGIN
A2 : out2 <= '1' AFTER 3 ns ;
A3 : out3 <= '0' AFTER 2 ns ;
   END BLOCK blk1 ;
```

【程序 3-14】

```
a1 : out1 <= '1' AFTER 3 ns ;
a2 : out2 <= '1' AFTER 3 ns ;
a3 : out3 <= '0' AFTER 2 ns ;
```

由于 VHDL 综合器不支持保护式 BLOCK 语句（GUARDED BLOCK），在此不拟讨论该语句的应用。但从综合的角度看，BLOCK 语句的存在也是毫无实际意义的，因为无论是否存在 BLOCK 语句结构，对于同一设计实体，综合后的逻辑功能是不会有任何变化的。在综合过程中，VHDL 综合器将略去所有的块语句。基于实用的观点，结构体中功能语句的划分最好使用元件例化（COMPONENT INSTANTIATION）的方式来完成。

§3.4 进程（PROCESS）

PROCESS 概念产生于软件语言，但在 VHDL 中，PROCESS 结构则是最具特色的语句，它的运行方式与软件语言中的 PROCESS 也完全不同，这是读者需要特别注意的。

PROCESS 语句结构包含了一个代表着设计实体中部分逻辑行为的、独立的顺序语句描述的进程。与并行语句的同时执行方式不同，顺序语句可以根据设计者的要求，利用顺序可控的语句，完成逐条执行的功能。顺序语句与 C 或 PASCAL 等软件编程语言中语句功能是相类似的，即语句运行的顺序是同程序语句书写的顺序相一致的。一个结构体中可以有多个并行运行的进程结构，而每一个进程的内部结构却是由一系列顺序语句来构成。

需要注意的是，在 VHDL 中，所谓顺序仅仅是指语句按序执行上的顺序性，但这并不意味着 PROCESS 语句结构所对应的硬件逻辑行为也具有相同的顺序性。PROCESS 结构中的顺序语句，及其所谓的顺序执行过程只是相对于计算机中的软件行为仿真的模拟过程而言的，这个过程与硬件结构中实现的对应的逻辑行为是不相同的。PROCESS 结构中既可以有时序逻辑的描述，也可以有组合逻辑的描述，它们都可以用顺序语句来表达。然而，硬件中的组合逻辑具有最典型的并行逻辑功能，而硬件中的时序逻辑也并非都是以顺序方式工作的。

1. PROCESS 语句格式

PROCESS 语句的表达格式如下：

```
[进程标号: ] PROCESS [ ( 敏感信号参数表  ) ] [IS]
[进程说明部分]
 BEGIN
  顺序描述语句
END PROCESS [进程标号];
```

每一个 PROCESS 语句结构可以赋予一个进程标号，但这个标号不是必需的。进程说明部分定义该进程所需的局部数据环境。

顺序描述语句部分是一段顺序执行的语句，描述该进程的行为。PROCESS 中规定了每个进程语句在当它的某个敏感信号（由敏感信号参量表列出）的值改变时都必须立即完成某一功能行为，这个行为由进程语句中的顺序语句定义，行为的结果可以赋给信号，并通过信号被其它的 PROCESS 或 BLOCK 读取或赋值。当进程中定义的任一敏感信号发生

更新时，由顺序语句定义的行为就要重复执行一次，当进程中最后一个语句执行完成后，执行过程将返回到进程的第一个语句，以等待下一次敏感信号变化。如此循环往复以至无限。但当遇到 WAIT 语句时，执行过程将被有条件地终止，即所谓的挂起（Suspention）。

一个结构体中可以含有多个 PROCESS 结构，每一 PROCESS 结构对于其敏感信号参数表中定义的任一敏感参量的变化，每个进程可以在任何时刻被激活或者称为启动。而在一结构体中，所有被激活的进程都是并行运行的，这就是为什么 PROCESS 结构本身是并行语句的道理。

PROCESS 语句必须以语句“END PROCESS [进程标号];”结尾，对于目前常用的综合器来说，其中进程标号不是必须的，敏感表旁的[IS]也不是必须的。

2. PROCESS 组成

如上所述，PROCESS 语句结构是由三个部分组成的，即进程说明部分、顺序描述语句部分和敏感信号参数表。

(1) 进程说明部分主要定义一些局部量，可包括数据类型、常数、变量、属性、子程序等。但需注意，在进程说明部分中不允许定义信号和共享变量。

(2) 顺序描述语句部分可分为赋值语句、进程启动语句、子程序调用语句、顺序描述语句和进程跳出语句等，它们包括：

- 信号赋值语句：即在进程中将计算或处理的结果向信号（SIGNAL）赋值。
- 变量赋值语句：即在进程中以变量（VARIABLE）的形式存储计算的中间值。
- 进程启动语句：当 PROCESS 的敏感信号参数表中没有列出任何敏感量时，进程的启动只能通过进程启动语句 WAIT 语句。这时可以利用 WAIT 语句监视信号的变化情况，以便决定是否启动进程。WAIT 语句可以看成是一种隐式的敏感信号表。
- 子程序调用语句：对已定义的过程和函数进行调用，并参与计算。
- 顺序描述语句：包括 IF 语句、CASE 语句、LOOP 语句、NULL 语句等。
- 进程跳出语句：包括 NEXT 语句、EXIT 语句，用于控制进程的运行方向。

(3) 敏感信号参数表需列出用于启动本进程可读入的信号名（当有 WAIT 语句时例外）。

程序 3-15 是一个含有进程的结构体，进程标号是 p1（进程标号不是必须的），进程的敏感信号参数表中未列出敏感信号，所以进程的启动需靠 WAIT 语句。在此，信号 clock 即为该进程的敏感信号。每当出现一个时钟脉冲 clock 时，即进入 WAIT 语句以下的顺序语句执行进程中，且当 driver 为高电平时进入 CASE 语句结构。

【程序 3-15】

```
ARCHITECURE  s_mode  OF  stat  IS
 BEGIN
  p1:  PROCESS
    BEGIN
     WAIT UNTIL clock  ;            -- 等待 clock 激活进程
     IF  (driver = '1' )  THEN
       CASE  output  IS
         WHEN  s1 =>  output <=  s2 ;
```

```
            WHEN  s2 =>  output <=  s3 ;
            WHEN  s3 =>  output <=  s4 ;
            WHEN  s4 =>  output <=  s1 ;
          END CASE ;
       END IF ;
     END PROCESS  p1 ;
END  ARCHITECURE  s_mode ;
```

例 3-16 是一个 4 位二进制加法计数器结构体内的逻辑描述，该结构体中的进程含有 IF 语句，进程定义了三个敏感信号 clk、clear、stop。当其中任何一个信号改变时，都将启动进程的运行。信号 cnt4 被综合器用寄存器来实现。

该计数器除了有时钟输入信号 clk 外，还设置了计数清零信号 clear 和计数使能信号 stop，进程都将它们列为敏感信号。

【程序 3-16】

```
SIGNAL cnt4 : INTEGER RANGE 0 TO 15 ;    -- 注意 cnt4 的数据类型
   ...
PROCESS (clk, clear, stop)
BEGIN
   IF clear = '0' THEN
      cnt4 <= 0 ;
   ELSIF clk'EVENT AND clk = '1' THEN    --如果遇到时钟上升沿，则…
      IF stop = '0' THEN                 --如果 stop 为低电平，则进行
         cnt4 <= cnt4 + 1 ;              --加法计数，否则停止计数
      END IF ;
    END IF ;
END PROCESS ;
```

3. 进程要点

从设计者的认识角度看，VHDL 程序与普通软件语言构成的程序有很大的不同，普通软件语言中的语句的执行方式和功能实现十分具体和直观，编程中，几乎可以立即作出判断。但 VHLD 程序，特别是进程结构，设计者应当从三个方面去判断它的功能和执行情况：

（1）基于 CPU 的纯软件的行为仿真运行方式；

（2）基于 VHDL 综合器的综合结果所可能实现的运行方式；

（3）基于最终实现的硬件电路的运行方式。

其它语句相比，进程语句结构具有更多的特点，对进程的认识和进行进程设计需要注意以下几方面的问题：

(1) 在同一结构体中的任一进程是一个独立的无限循环程序结构，但进程中却不必放置诸如软件语言中的返回语句，它的返回是自动的。进程只有两种运行状态，即执行状态和等待状态。进程是否进入执行状态，取决于是否满足特定的条件，如敏感变量是否发生变化。如果满足条件，即进入执行状态，当遇到 END PROCESS 语句后即停止执行，自动返回到起始语句 PROCESS，进入等待状态。

(2) 必须注意，PROCESS 中的顺序语句的执行方式与通常的软件语言中的语句的顺序执行方式有很大的不同。软件语言中每一条语句的执行是按 CPU 的机器周期的节拍顺

序执行的，每一条语句的执行的时间与 CPU 的工作方式、工作晶振的频率、机器周期及指令周期的长短有密切的关系；但在 PROCESS 中，一个执行状态的运行周期，即从 PROCESS 的启动执行到遇到 END PROCESS 为止所花的时间与任何外部因素都无关（从综合结果来看），甚至与 PROCESS 语法结构中的顺序语句的多少都没有关系，其执行时间从行为仿真的角度看只有一个 VHDL 模拟器的最小分辨时间，即一个 δ 时间；但从综合和硬件运行的角度看，其执行时间是 0，这与信号的传输延时无关，与被执行的语句的实现时间也无关。即在同一 PROCESS 中，10 条语句和 1000 条语句的执行时间是一样的。这就是为什么用进程的顺序语句方式也同样能描述全并行的逻辑工作方式的道理。

（3）虽然同一结构体中的不同进程是并行运行的，但同一进程中的逻辑描述语句则是顺序运行的，因而在进程中只能设置顺序语句。

（4）进程的激活必须由敏感信号表中定义的任一敏感信号的变化来启动，否则必须有一个显式的 WAIT 语句来激励。这就是说，进程既可以通过敏感信号的变化来启动，也可以由满足条件的 WAIT 语句而激活；反之，在遇到不满足条件的 WAIT 语句后进程将被挂起。因此，进程中必须定义显式或隐式的敏感信号。如果一个进程对一个信号集合总是敏感的，那么，我们可以使用敏感表来指定进程的敏感信号。但是，在一个使用了敏感表的进程（或者由该进程所调用的子程序）中不能含有任何等待语句。

（5）结构体中多个进程之所以能并行同步运行，一个很重要的原因是进程之间的通信是通过传递信号和共享变量值来实现的。所以相对于结构体来说，信号具有全局特性，它是进程间进行并行联系的重要途径。因此，在任一进程的进程说明部分不允许定义信号和共享变量（共享变量是 VHDL'93 增加的内容）。

（6）进程是 VHDL 重要的建模工具。与 BLOCK 语句不同的一个重要方面是，进程结构不但为综合器所支持，而且进程的建模方式将直接影响仿真和综合结果。

（7）进程有组合进程和时序进程两种类型，组合进程只产生组合电路，时序进程产生时序和相配合的组合电路，这两种类型的进程设计必须密切注意 VHDL 语句应用的特殊方面，这在多进程的状态机的设计中，各进程有明确分工。设计中，需要特别注意的是，组合进程中所有输入信号，包括赋值符号右边的所有信号和条件表达式中的所有信号，都必须包含于此进程的敏感信号表中！否则，当没有被包括在敏感信号表中的信号发生变化时，进程中的输出信号不能按照组合逻辑的要求得到即时的新的信号，VHDL 综合器将会给出错误判断，将误判为设计者有存储数据的意图，即判断为时序电路。这时综合器将会为对应的输出信号引入一个保存原值的锁存器，这样就打破了设计组合进程的初衷。在实际电路中，这类“组合进程”的运行速度、逻辑资源效率和工作可靠性都将受到不良影响。

时序进程必须是列入敏感表中某一时钟信号的同步逻辑，或同一时钟信号使结构体中的多个时序进程构成同步逻辑。当然，一个时序进程也可以利用另一进程（组合或时序进程）中产生的信号作为自己的时钟信号。引入时序元件的详细方法可参阅第 9、10 章。

§ 3.5 子程序(SUBPROGRAM)

子程序是一个 VHDL 程序模块，这个模块是利用顺序语句来定义和完成算法的，因此只能使用顺序语句，这一点与进程十分相似。所不同的是，子程序不能像进程那样可以从本结构体的其它块或进程结构中直接读取信号值或者向信号赋值。此外，VHDL 子程序与其它软件语言程序中的子程序的应用目的是相似的，即能更有效地完成重复性的计算工作。子程序的使用方式只能通过子程序调用及与子程序的界面端口进行通信。子程序的应用与元件例化（元件调用）是不同的，如果在一个设计实体或另一个子程序中调用子程序后，并不像元件例化那样会产生一个新的设计层次。

子程序可以在 VHDL 程序的 3 个不同位置进行定义，即在程序包、结构体和进程中定义。但由于只有在程序包中定义的子程序可被几个不同的设计所调用，所以一般应该将子程序放在程序包中。

VHDL 子程序具有可重载性的特点，即允许有许多重名的子程序，但这些子程序的参数类型及返回值数据类型是不同的。子程序的可重载性是一个非常有用的特性。

子程序有两种类型，即过程（PROCEDURE）和函数（FUNCTION）。

过程的调用可通过其界面提供多个返回值，或不提供任何值，而函数只能返回一个值。在函数入口中，所有参数都是输入参数，而过程有输入参数、输出参数和双向参数。过程一般被看作一种语句结构，常在结构体或进程中以分散的形式存在，而函数通常是表达式的一部分，常在赋值语句或表达式中使用。过程可以单独存在，其行为类似于进程，而函数通常作为语句的一部分被调用（在第 5、6 章中，对子程序的应用有更具体的说明）。

在实用中必须注意，综合后的子程序将映射于目标芯片中的一个相应的电路模块，且每一次调用都将在硬件结构中产生对应于具有相同结构的不同的模块，这一点与在普通的软件中调用子程序有很大的不同。在 PC 机或单片机软件程序执行中（包括 VHDL 的行为仿真），无论对程序中的子程序调用多少次，都不会发生计算机资源，如存储资源不够用的情况，但在面向 VHDL 的综合中，每调用一次子程序都意味着增加了一个硬件电路模块。因此，在实用中，要密切关注和严格控制子程序的调用次数。

3.5.1 函数（FUNCTION）

在 VHDL 中有多种函数形式，如用于不同目的的用户自定义函数和在库中现成的具有专用功能的预定义函数。例如转换函数和决断函数。转换函数用于从一种数据类型到另一种数据类型的转换，如在元件例化语句中利用转换函数可允许不同数据类型的信号和端口间进行映射；决断函数用于在多驱动信号时解决信号竞争问题。

函数的语言表达格式如下：

```
FUNCTION 函数名（参数表） RETURN  数据类型          --函数首
```

```
FUNCTION  函数名（参数表）RETURN  数据类型 IS        -- 函数体
   [ 说明部分 ]
   BEGIN
   顺序语句 ;
END FUNCTION  函数名;
```

一般地，函数定义应由两部分组成，即函数首和函数体，在进程或结构体中不必定义函数首，而在程序包中必须定义函数首。

1. 函数首

函数首是由函数名、参数表和返回值的数据类型三部分组成的，如果将所定义的函数组织成程序包入库的话，定义函数首是必需的，这时的函数首就相当于一个入库货物名称与货物位置表，入库的是函数体。函数首的名称即为函数的名称，需放在关键词 FUNCTION 之后，此名称可以是普通的标识符，也可以是运算符，运算符必须加上双引号，这就是所谓的运算符重载。运算符重载就是对 VHDL 中现存的运算符进行重新定义，以获得新的功能。新功能的定义是靠函数体来完成的，函数的参数表是用来定义输出值的，所以不必以显式表示参数的方向，函数参量可以是信号或常数，参数名需放在关键词 CONSTANT 或 SIGNAL 之后。如果没有特别说明，则参数被默认为常数。如果要将一个已编制好的函数并入程序包，函数首必须放在程序包的说明部分，而函数体需放在程序包的包体内。如果只是在一个结构体中定义并调用函数，则仅需函数体即可。由此可见，函数首的作用只是作为程序包的有关此函数的一个接口界面。有关的示例如下：

【程序 3-17】

```
PACKAGE  packexp IS                                --定义程序包
  FUNCTION  max( a,b : IN STD_LOGIC_VECTOR)  --定义函数首
    RETURN STD_LOGIC_VECTOR ;
  FUNCTION  func1 ( a ,b ,c : REAL )          --定义函数首
    RETURN REAL ;
  FUNCTION  "*"  ( a ,b : INTEGER )           --定义函数首
    RETURN INTEGER ;
  FUNCTION  as2  (SIGNAL  in1 ,in2 : REAL )   --定义函数首
    RETURN REAL ;
END;

PACKAGE  BODY packexp IS
  FUNCTION  max( a,b : IN STD_LOGIC_VECTOR)   --定义函数体
    RETURN STD_LOGIC_VECTOR IS
  BEGIN
    IF a > b THEN RETURN  a;
      ELSE        RETURN  b;
    END IF;
  END FUNCTION max;          --结束 FUNCTION 语句
END;                         --结束 PACKAGE  BODY 语句


 -- 函数应用实例
```

```
...
USE WORK. packexp.ALL;
ENTITY  axamp IS
  PORT(...);
END;
ARCHITECTURE bhv OF axamp IS
  BEGIN
   ...
   out1 <=  max(dat1,dat2);  --用在赋值语句中的并行函数调用语句
  PROCESS(dat3,dat4)
  BEGIN
   out2 <= max(dat3,dat4);      --顺序函数调用语句
  END PROCESS;
   ...
 END;
```

程序 3-17 有 4 个不同的函数首，它们都放在程序包 packexp 的说明部分。

第 1 个函数中的参量 a、b 的数据类型是标准位矢量类型，返回值是 a、b 中的最大值，其数据类型也是标准位矢量类型。

第 2 个函数中的参量 a、b、c 的数据类型都是实数类型，返回值也是实数类型。

第 3 个函数定义了一种新的乘法算符，即通过用此函数定义的算符"*"可以进行两个整数间的乘法，且返回值也是整数。值得注意的是，这个函数的函数名用的是以双引号相间的乘法算符，对于其它算符的重载定义也必须加双引号，如"+"。

最后一个函数定义的输入参量是信号。书写格式上，在函数名后的括号中先写上参量目标类型 SIGNAL ，以表示 in1 和 in2 是两个信号，最后写上此两个信号的数据类型是实数 REAL，返回值也是实数类型。

2．函数体

函数体包含一个对数据类型、常数、变量等的局部说明，以及用以完成规定算法或转换的顺序语句部分。一旦函数被调用，就将执行这部分语句。

在函数体结尾需以关键词 END FUNCTION 以及函数名结尾。

以上的程序 3-17 是一个将函数定义于程序包，及实际应用的实例，程序中段，有一个对函数 max 的函数体的定义实例，其中以顺序语句描述了此函数的功能；下段给出了一个调用此函数的应用实例。

以下的程序 3-18 在一个结构体内定义了一个完成某种算法的函数，并在进程 PROCESS 中调用了此函数，这个函数没有函数首。在进程中，输入端口信号位矢 a 被列为敏感信号，当 a 的 3 个位输入元素 a(0)、a(1)和 a(2)中的任何一位有变化时，将启动对函数 sam 的调用，并将函数的返回值赋给 m 输出。

【程序 3-18】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY func IS
  PORT  （ a :   IN STD_LOGIC_VECTOR (0 to 2 ) ;
           m :  OUT STD_LOGIC_VECTOR (0 to 2 )   ）;
```

```
END ENTITY func ;
ARCHITECTURE demo OF func IS
FUNCTION sam(x ,y ,z : STD_LOGIC) RETURN STD_LOGIC IS
BEGIN
  RETURN ( x AND y ) OR y  ;
END FUNCTION sam ;
BEGIN
  PROCESS ( a )
  BEGIN
   m(0) <= sam( a(0),  a(1),  a(2) ) ;
   m(1) <= sam( a(2),  a(0),  a(1) ) ;
   m(2) <= sam( a(1),  a(2),  a(0) ) ;
  END PROCESS ;
END ARCHITECTURE demo ;
```

程序 3-19 给出的函数体是通过满足某种条件来完成算法的，它返回的值也是位矢。注意，MAX+PLUSII 不支持 IF 或 CASE 语句中的 RETURN 语句，所以不支持程序 3-19 给出的语法格式，这是 MAX+PLUSII 的缺陷。

【程序 3-19】

```
FUNCTION trans ( value : IN BIT_VECTOR (0 TO 1) ) ;
   RETURN BIT_VECTOR IS
  BEGIN
  CASE value IS
  WHEN "0000" => RETURN "1100" ;
  WHEN "0101" => RETURN "0001" ;
  WHEN OTHERS => RETURN "1111" ;
  END CASE ;
END FUNCTION trans ;
```

在参数说明部分的“IN”不是必需的(第 5 章中将进一步讨论函数的使用方法)。

3.5.2 重载函数（OVERLOADED FUNCTION）

VHDL 允许以相同的函数名定义函数，但要求函数中定义的操作数具有不同的数据类型，以便调用时用以分辨不同功能的同名函数。即同样名称的函数可以用不同的数据类型作为此函数的参数定义多次，以此定义的函数称为重载函数。函数还可以允许用任意位矢长度来调用。程序 3-20 是一个比较完整的重载函数 max 的定义和调用的实例：

【程序 3-20】

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
PACKAGE  packexp IS                              --定义程序包
 FUNCTION  max( a,b : IN STD_LOGIC_VECTOR)   --定义函数首
   RETURN STD_LOGIC_VECTOR ;
 FUNCTION  max( a,b : IN BIT_VECTOR)   --定义函数首
```

```
  RETURN BIT_VECTOR ;
 FUNCTION  max( a,b : IN INTEGER )     --定义函数首
  RETURN INTEGER ;
 END;

PACKAGE BODY packexp IS
 FUNCTION  max( a,b : IN STD_LOGIC_VECTOR)   --定义函数体
  RETURN STD_LOGIC_VECTOR IS
 BEGIN
  IF a > b THEN RETURN  a;
    ELSE        RETURN  b;    END IF;
 END FUNCTION max;        --结束 FUNCTION 语句

 FUNCTION  max( a,b : IN INTEGER)   --定义函数体
  RETURN INTEGER IS
 BEGIN
  IF a > b THEN RETURN  a;
    ELSE        RETURN  b;    END IF;
 END FUNCTION max;        --结束 FUNCTION 语句

 FUNCTION  max( a,b : IN BIT_VECTOR)   --定义函数体
  RETURN BIT_VECTOR IS
 BEGIN
  IF a > b THEN RETURN  a;
    ELSE        RETURN  b;    END IF;
 END FUNCTION max;        --结束 FUNCTION 语句
END;                      --结束 PACKAGE  BODY 语句
```

-- 以下是调用重载函数 max 的程序：

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
 USE WORK.packexp.ALL;
ENTITY  axamp IS
  PORT(a1,b1 : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       a2,b2 : IN BIT_VECTOR(4 DOWNTO 0);
       a3,b3 : IN INTEGER 0 TO 15;
          c1 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
          c2 : OUT BIT_VECTOR(4 DOWNTO 0);
          c3 : OUT INTEGER 0 TO 15);
 END;
ARCHITECTURE bhv OF axamp IS
  BEGIN
  c1 <=  max(a1,b1); --对函数 max( a,b : IN STD_LOGIC_VECTOR)的调用
  c2 <=  max(a2,b2); --对函数 max( a,b : IN BIT_VECTOR) 的调用
  c3 <=  max(a3,b3); --对函数 max( a,b : IN INTEGER)  的调用
END;
```

作为强类型语言，VHDL 不允许不同数据类型的操作数间进行直接操作或运算。为此，在具有不同数据类型操作数构成的同名函数中，可定义以运算符重载式的重载函数，这种函数为不同数据类型间的运算带来极大的方便。程序 3-21 中以加号“+”为函数名的函数即为运算符重载函数。VHDL 的 IEEE 库中的 STD_LOGIC_UNSIGNED 程序包中预定义的操作符如"+"、"-"、"*"、"="、">="、"<="、">"、"<"、"/="、"AND"和"MOD"等，对相应的数据类型 INTEGRE、STD_LOGIC 和 STD_LOGIC_VECTOR 的操作作了重载，赋予了新的数据类型操作功能，即通过重新定义运算符的方式，允许被重载的运算符能够对新的数据类型进行操作，或者允许不同的数据类型之间用此运算符进行运算。程序 3-21 给出了一个 Synopsys 公司的程序包 STD_LOGIC_UNSIGNED 中的部分函数结构。示例没有把全部内容列出。在程序包 STD_LOGIC_UNSIGNED 的说明部分只列出了四个函数的函数首；在程序包体部分只列出了对应的部分内容，程序包体部分的 UNSIGNED（）函数是从 IEEE.STD_LOGIC_ARITH 库中调用的，在程序包体中的最大整型数检出函数 MAXIUM 只有函数体，没有函数首，这是因为它只在程序包体内调用。

【程序 3-21】

```
LIBRARY IEEE ;                                          -- 程序包首
USE IEEE.std_logic_1164.all ;
USE IEEE.std_logic_arith.all ;
PACKAGE STD_LOGIC_UNSIGNED is
function "+" (L : STD_LOGIC_VECTOR ; R : INTEGER)
                return STD_LOGIC_VECTOR ;
function "+" (L : INTEGER;  R : STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR ;
function "+" (L : STD_LOGIC_VECTOR ; R : STD_LOGIC )
return STD_LOGIC_VECTOR ;
function SHR (ARG : STD_LOGIC_VECTOR ;
  COUNT : STD_LOGIC_VECTOR ) return STD_LOGIC_VECTOR ;
...
end STD_LOGIC_UNSIGNED  ;

LIBRARY IEEE ;                                          --  程序包体
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
package body STD_LOGIC_UNSIGNED is
function maximum (L, R :  INTEGER) return  INTEGER  is
begin
   if L > R then
      return L;
   else
      return R;
   end if;
end;
function "+" (L : STD_LOGIC_VECTOR ;  R : INTEGER)
return  STD_LOGIC_VECTOR  is
Variable result : STD_LOGIC_VECTOR (L'range) ;
Begin
  result := UNSIGNED(L) + R ;
```

```
  return std_logic_vector(result) ;
end ;
...
end STD_LOGIC_UNSIGNED ;
```

通过此例，读者不但可以看到在程序包中完整的函数置位形式，而且还将注意到，在函数首的三个函数的函数名都是同名的，即都是以加法运算符“+”作为函数名，以这种方式定义函数即所谓运算符重载。对运算符重载，即对运算符重新定义的函数称重载函数。

实用中，如果已用“USE”语句打开了程序包 STD_LOGIC_UNSIGNED，这时，如果设计实体中有一个 STD_LOGIC_VECTOR 位矢和一个整数相加，程序就会自动调用第一个函数，并返回位矢类型的值；若有一个位矢与 STD_LOGIC 数据类型的数相加，则调用第三个函数，并以位矢类型的值返回。

有了重载函数，4 位二进制加法计数器就可以用以下的 VHDL 程序实现了。试比较程序 3-22 与程序 3-16 中操作数数据类型方面的不同之处。

【程序 3-22】

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
USE IEEE.STD_LOGIC_UNSIGNED.ALL ;     -- 注意此程序包的功能!
ENTITY cnt4 IS
   PORT
         Clk : IN STD_LOGIC ;
          q : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0)
         );
END cnt4;
ARCHITECTURE one OF cnt4 IS
BEGIN
   PROCESS ( clk )
   BEGIN
      IF  clk'EVENT AND clk = '1'  THEN
          IF q=15 THEN     -- 这里，程序自动调用了等号“=”的重载函数
            q <= "0000" ;
          ELSE
            q <= q + 1 ;   -- 这里，程序自动调用了加号“+”的重载函数
          END IF ;
      END IF ;
  END PROCESS ;
END one ;
```

此例中，式“q = 15”等号两边的数据类型是不一样的，q 的数据类型是位矢量类型（位的数组类型），而“15”属于整数类型；式“q<=q + 1”中“+”两边的数据类型也不一样，“1”也是整数类型。之所以不同类型的操作数可以在一起作用，全得益于利用了语句：

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL
```

打开了运算符重载函数的程序包。

3.5.3 过程（PROCEDURE）

VHDL 中，子程序的另外一种形式是过程 PROCEDURE，过程的语句格式是：

```
PROCEDURE 过程名（参数表）                    -- 过程首

PROCEDURE 过程名（参数表） IS
    [说明部分]
    BIGIN                                    -- 过程体
      顺序语句;
     END PROCEDURE 过程名;
```

与函数一样，过程也由两部分组成，即由过程首和过程体构成，过程首也不是必需的，过程体可以独立存在和使用。即在进程或结构体中不必定义过程首，而在程序包中必须定义过程首。

1. 过程首

过程首由过程名和参数表组成。参数表可以对常数、变量和信号三类数据对象目标作出说明，并用关键词 IN、OUT 和 INOUT 定义这些参数的工作模式，即信息的流向。如果没有指定模式，则默认为 IN。以下是三个过程首的定义示例：

【程序 3-23】

```
PROCEDURE  pro1  (VARIABLE  a, b :  INOUT REAL) ;
PROCEDURE  pro2  (CONSTANT  a1 :  IN INTEGER ;
                  VARIABLE  b1 :  OUT INTEGER ) ;
PROCEDURE  pro3  (SIGNAL sig :  INOUT BIT) ;
```

过程 pro1 定义了两个实数双向变量 a 和 b ；过程 pro2 定义了两个参量。第一个是常数，它的数据类型为整数，流向模式是 IN，第二个参量是变量，信号模式和数据类型分别是 OUT 和整数；过程 pro3 中只定义了一个信号参量，即 sig，它的流向模式是双向 INOUT，数据类型是 BIT。一般地，可在参量表中定义三种流向模式，即 IN、OUT 和 INOUT。如果只定义了 IN 模式而未定义目标参量类型，则默认为常量；若只定义了 INOUT 或 OUT，则默认目标参量类型是变量。

2. 过程体

过程体是由顺序语句组成的，过程的调用即启动了对过程体的顺序语句的执行。与函数一样，过程体中的说明部分只是局部的，其中的各种定义只能适用于过程体内部。过程体的顺序语句部分可以包含任何顺序执行的语句，包括 WAIT 语句。但需注意，如果一个过程是在进程中调用的，且这个进程已列出了敏感参量表，则不能在此过程中使用 WAIT 语句。

在不同的调用环境中，可以有两种不同的语句方式对过程进行调用，即顺序语句方式或并行语句方式。对于前者，在一般的顺序语句自然执行过程中，一个过程被执行，则属于顺序语句方式，因为这时它只相当于一条顺序语句的执行；对于后

者，一个过程相当于一个小的进程，当这个过程处于并行语句环境中时，其过程体中定义的任一 IN 或 INOUT 的目标参量（即数据对象：变量、信号、常数）发生改变时，将启动过程的调用，这时的调用是属于并行语句方式的。过程与函数一样可以重复调用或嵌套式调用。综合器一般不支持含有 Wait 语句的过程。以下是两个过程体的使用示例：

【程序 3-24】

```
PROCEDURE  prg1(VARIABLE value:INOUT BIT_VECTOR(0 TO 7)) IS
  BEGIN
   CASE value IS
    WHEN "0000" => value: "0101" ;
    WHEN "0101" => value: "0000" ;
    WHEN OTHERS => value: "1111" ;
   END CASE ;
    END PROCEDURE  prg1 ;
```

这个过程对具有双向模式变量的值 value 作了一个数据转换运算。

【程序 3-25】

```
PROCEDURE  comp ( a, r : IN REAL;
                     m : IN INTEGER ;
                  v1, v2: OUT REAL) IS
VARIABLE cnt : INTEGER ;
BEGIN
     v1 := 1.6 * a ;                      -- 赋初始值
     v2 := 1.0 ;                          -- 赋初始值
Q1 :  FOR  cnt  IN  1 TO m  LOOP
      v2 :=  v2 * v1 ;
   EXIT  Q1  WHEN  v2 > v1;               -- 当 v2 > v1，跳出循环 LOOP
 END LOOP Q1
    ASSERT (v2 < v1 )
        REPORT  "OUT OF RANGE"            -- 输出错误报告
           SEVERITY  ERROR ;
END PROCEDURE  comp  ;
```

在以上过程 comp 的参量表中，定义 a 和 r 为输入模式，数据类型为实数；m 为输入模式，数据类型为整数。这三个参量都没有以显式定义它们的目标参量类型，显然它们的默认类型都是常数。由于 v2、v1 定义为输入模式的实数，因此默认类型是变量。在过程 comp 的 LOOP 语句中，对 v2 进行循环计算直到 v2 大于 r， EXIT 语句中断运算，并由 REPORT 语句给出错误报告。

3.5.4 重载过程（OVERLOADED PROCEDURE）

两个或两个以上有相同的过程名和互不相同的参数数量及数据类型的过程称为重载过程。十分类似于重载函数，对于重载过程，也是靠参量类型来辨别究竟调用哪一个过程。

【程序 3-26】

```
PROCEDURE  calcu (  v1, v2 : IN REAL ;
                SIGNAL out1 : INOUT INTEGER) ;
PROCEDURE  calcu (  v1, v2 : IN INTEGER ;
                SIGNAL out1 : INOUT REAL) ;
...
calcu (20.15, 1.42, signl) ;     -- 调用第一个重载过程 calcu
calcu (23,  320, sign2 ) ;       -- 调用第二个重载过程 calcu
...
```

此例中定义了两个重载过程，它们的过程名、参量数目及各参量的模式是相同的，但参量的数据类型是不同的。第一个过程中定义的两个输入参量 v1 和 v2 为实数型常数，out1 为 INOUT 模式的整数信号；而第二个过程中 v1、v2 则为整数常数，out1 为实数信号。所以在下面过程调用中将首先调用第一个过程。

如前所述，在过程结构中的语句是顺序执行的，调用者在调用过程前应先将初始值传递给过程的输入参数，一旦调用，即启动过程语句按顺序自上而下执行过程中的语句，执行结束后，将输出值返回到调用者的“OUT”和“INOUT”所定义的变量或信号中。

另外，从程序 3-26 可见，过程的调用方式与函数完全不同。函数的调用中，是将所定义的函数作为语句中的一个因子，如一个操作数或一个赋值数据对象或信号等，而过程的调用，是将所定义的过程名作为一条语句来执行。

§ 3.6 库（LIBRARY）

在利用 VHDL 进行工程设计中，为了提高设计效率以及使设计遵循某些统一的语言标准或数据格式，有必要将一些有用的信息汇集在一个或几个库中以供调用。这些信息可以是预先定义好的数据类型、子程序等设计单元的集合体（程序包），或预先设计好的各种设计实体（元件库程序包）。因此，可以把库看成是一种用来存储预先完成的程序包、数据集合体和元件的仓库。如果要在一项 VHDL 设计中用到某一程序包，就必须在这项设计中预先打开这个程序包，使此设计能随时使用这一程序包中的内容。在综合过程中，每当综合器在较高层次的 VHDL 源文件中遇到库语言，就将随库指定的源文件读入，并参与综合。这就是说，在综合过程中，所要调用的库必须以 VHDL 源文件的方式存在，并能使综合器随时读入使用。为此必须在这一设计实体前使用库语句和 USE 语句（USE 语句将在后面介绍）。一般地，在 VHDL 程序中被声明打开的库和程序包，对于本项设计称为是可视的，那么这些库中的内容就可以被设计项目所调用。有些库被 IEEE 认可，成为 IEEE 库，IEEE 库存放了 IEEE 标准 1076 中标准设计单元，如 Synopsys 公司的 STD_LOGIC_UNSIGNED 程序包等。

通常，库中放置不同数量的程序包，而程序包中又可放置不同数量的少子程序；子程序中又含有函数、过程、设计实体（元件）等基础设计单元。

VHDL 语言的库分为两类：一类是设计库，如在具体设计项目中设定的目录所对应的

WORK 库，另一类是资源库，资源库是常规元件和标准模块存放的库，如 IEEE 库。设计库对当前项目是默认可视的，无需用 LIBRARY 和 USE 等语句以显式声明。

库（LIBRARY）的语句格式如下：

```
LIBRARY  库名;
```

这一语句即相当于为其后的设计实体打开了以此库名命名的库，以便设计实体可以利用其中的程序包。如语句“ LIBRARY IEEE ;”表示打开 IEEE 库。

1. 库的种类

VHDL 程序设计中常用的库有以下几种：

- IEEE 库

IEEE 库是 VHDL 设计中最为常见的库，它包含有 IEEE 标准的程序包和其它一些支持工业标准的程序包。IEEE 库中的标准程序包主要包括 STD_LOGIC_1164，NUMERIC_BIT 和 NUMERIC_STD 等程序包。其中的 STD_LOGIC_1164 是最重要和最常用的程序包，大部分基于数字系统设计的程序包都是以此程序包中设定的标准为基础的。

此外，还有一些程序包虽非 IEEE 标准，但由于其已成事实上的工业标准，也都并入了 IEEE 库。这些程序包中，最常用的是 Synopsys 公司的 STD_LOGIC_ARITH、STD_LOGIC_SIGNED 和 STD_LOGIC_UNSIGNED 程序包，目前流行于我国的大多数 EDA 工具都支持 Synopsys 公司的程序包。一般基于大规模可编程逻辑器件的数字系统设计，IEEE 库中的四个程序包 STD_LOGIC_1164、STD_LOGIC_ARITH、STD_LOGIC_SIGNED 和 STD_LOGIC_UNSIGNED 已足够使用。另外需要注意的是，在 IEEE 库中符合 IEEE 标准的程序包并非符合 VHDL 语言标准，如 STD_LOGIC_1164 程序包。因此在使用 VHDL 设计实体的前面必须以显式表达出来。

- STD 库

VHDL 语言标准定义了两个标准程序包，即 STANDARD 和 TEXTIO 程序包（文件输入/输出程序包），它们都被收入在 STD 库中，只要在 VHDL 应用环境中，即可随时调用这两个程序包中的所有内容，即在编译和综合过程中，VHDL 的每一项设计都自动地将其包含进去了。由于 STD 库符合 VHDL 语言标准，在应用中不必如 IEEE 库那样以显式表达出来，如在程序中，以下的库使用语句是不必要的。

```
LIBRARY STD ;
USE STD.STANDARD.ALL ;
```

- WORK 库

WORK 库是用户的 VHDL 设计的现行工作库，用于存放用户设计和定义的一些设计单元和程序包，因而是用户的临时仓库，用户设计项目的成品、半成品模块，以及先期已设计好的元件都放在其中。WORK 库自动满足 VHDL 语言标准，在实际调用中，也不必以显式预先说明。基于 VHDL 所要求的 WORK 库的基本概念，在 PC 机或工作站上利用 VHDL 进行项目设计，不允许在根目录下进行，而是必须为此设定一个目录，用于保存所有此项

目的设计文件，VHDL 综合器将此目录默认为 WORK 库。但必须注意，工作库并不是这个目录的目录名，而是一个逻辑名。综合器将指示器指向该目录的路径。VHDL 标准规定工作库总是可见的，因此，不必在 VHDL 程序中明确指定。

- VITAL 库

使用 VITAL 库，可以提高 VHDL 门级时序模拟的精度，因而只在 VHDL 仿真器中使用。库中包含时序程序包 VITAL_TIMING 和 VITAL_PRIMITIVES。VITAL 程序包已经成为 IEEE 标准，在当前的 VHDL 仿真器的库中，VITAL 库中的程序包都已经并到 IEEE 库中。实际上，由于各 FPGA/CPLD 生产厂商的适配工具（如 ispEXPERT Compiler，参见第 12 章）都能为各自的芯片生成带时序信息的 VHDL 门级网表，用 VHDL 仿真器仿真该网表可以得到非常精确的时序仿真结果。因此，基于实用的观点，在 FPGA/CPLD 设计开发过程中，一般并不需要 VITAL 库中的程序包。

除了以上提到的库外，EDA 工具开发商为了 FPGA/CPLD 开发设计上的方便，都有自己的扩展库和相应的程序包，如 DATAIO 公司的 GENERICS 库、DATAIO 库等，以及上面提到的 Synopsys 公司的一些库。

在 VHDL 设计中，有的 EDA 工具将一些程序包和设计单元放在一个目录下，而将此目录名，如“WORK”，作为库名，如 Synplicity 公司的 Synplify (详细用法可参见第 12 章)。有的 EDA 工具是通过配置语句结构来指定库和库中的程序包，这时的配置即成为一个设计实体中最顶层的设计单元。

此外，用户还可以自己定义一些库，将自己的设计内容或通过交流获得的程序包设计实体并入这些库中。

2．库的用法

在 VHDL 语言中，库的说明语句总是放在实体单元前面。这样，在设计实体内的语句就可以使用库中的数据和文件。由此可见，库的用处在于使设计者可以共享已经编译过的设计成果。VHDL 允许在一个设计实体中同时打开多个不同的库，但库之间必须是相互独立的。

程序 3-22 中最前面的三条语句：

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
USE IEEE.STD_LOGIC_UNSIGNED.ALL ;
```

表示打开 IEEE 库，再打开此库中的 STD_LOGIC_1164 程序包和 STD_LOGIC_UNSIGNED 程序包的所有内容。由此可见，在实际使用中，库是以程序包集合的方式存在的，具体调用的是程序包中的内容，因此对于任一 VHDL 设计，所需从库中调用的程序包在设计中应是可见的（可调出的），即以明确的语句表达方式加以定义，库语句指明库中的程序包以及包中的待调用文件。

对于必须以显式表达的库及其程序包的语言表达式应放在每一项设计实体最前面，成为这项设计的最高层次的设计单元。库语句一般必须与 USE 语句同用。库语句关键词 LIBRARY，指明所使用的库名。USE 语句指明库中的程序包。一旦说明了库和程序包，

整个设计实体都可进入访问或调用，但其作用范围仅限于所说明的设计实体。VHDL 要求一项含有多个设计实体的更大的系统中，每一个设计实体都必须有自己完整的库说明语句和 USE 语句。

USE 语句的使用将使所说明的程序包对本设计实体部分或全部开放，即是可视的。USE 语句的使用有两种常用格式：

```
USE 库名.程序包名.项目名 ;
USE 库名.程序包名.ALL ;
```

第一语句格式的作用是，向本设计实体开放指定库中的特定程序包内所选定的项目。

第二语句格式的作用是，向本设计实体开放指定库中的特定程序包内所有的内容。

合法的 USE 语句的使用方法是，将 USE 语句说明中所要开放的设计实体对象紧跟在 USE 语句之后。例如，语句

```
USE IEEE.STD_LOGIC_1164.ALL ;
```

表明打开 IEEE 库中的 STD_LOGIC_1164 程序包，并使程序包中所有的公共资源对于本语句后面的 VHDL 设计实体程序全部开放，即该语句后的程序可任意使用程序包中的公共资源。这里用到了关键词“ALL”，代表程序包中所有资源。

【程序 3-27】

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.STD_ULOGIC ;
USE IEEE.STD_LOGIC_1164.RISING_EDGE ;
```

此例中向当前设计实体开放了 STD_LOGIC_1164 程序包中的 RISING_EDGE 函数，但由于此函数需要用到数据类型 STD_ULOGIC，所以在上一条 USE 语句中开放了同一程序包中的这一数据类型。

§3.7 程序包（PACKAGE）

已在设计实体中定义的数据类型、子程序或数据对象对于其它设计实体是不可用的，或者说是不可见的。为了使已定义的常数、数据类型、元件调用说明以及子程序能被更多的 VHDL 设计实体方便地访问和共享，可以将它们收集在一个 VHDL 程序包中。多个程序包可以并入一个 VHDL 库中，使之适用于更一般的访问和调用范围。这一点对于大系统开发，多个或多组开发人员同步并行工作显得尤为重要。

程序包的内容主要由如下四种基本结构组成，因此一个程序包中至少应包含以下结构中的一种。

- 常数说明：在程序包中的常数说明结构主要用于预定义系统的宽度，如数据总线通道的宽度。
- VHDL 数据类型说明：主要用于在整个设计中通用的数据类型，例如通用的地址总线数据类型定义等（第 4 章将对数据类型作详细说明）。

• **元件定义：元件定义主要规定在 VHDL 设计中参与文件例化的文件（已完成的设计实体）对外的接口界面。**

• **子程序：并入程序包的子程序有利于在设计中任一处进行方便地调用。**

通常程序包中的内容应具有更大的适用面和良好的独立性，以供各种不同设计需求的调用，如 STD_LOGIC_1164 程序包定义的数据类型 STD_LOGIC 和 STD_LOGIC_VECTOR。一旦定义了一个程序包，各种独立的设计就能方便地调用。

定义程序包的一般语句结构如下：

```
PACKAGE  程序包名  IS                --  程序包首
    程序包首说明部分
END  程序包名;

PACKAGE BODY  程序包名  IS             --  程序包体
程序包体说明部分以及包体内
END  程序包名;
```

程序包的结构由程序包的说明部分即程序包首和程序包的内容部分即程序包体两部分组成。一个完整的程序包中，程序包首的程序包名与程序包体的程序包名是同一个名字。如例 3-21 所示的程序包 STD_LOGIC_UNSIGNED 是程序包组成结构的一个很好的示例。

1. 程序包首

程序包首的说明部分可收集多个不同的 VHDL 设计所需的公共信息，其中包括数据类型说明、信号说明、子程序说明及元件说明等。所有这些信息虽然也可以在每一个设计实体中进行逐一单独的定义和说明，但如果将这些经常用到的、并具有一般性的说明定义放在程序包中供随时调用，显然可以提高设计的效率和程序的可读性。

程序包结构中，程序包体并非总是是必须的，程序包首也可以独立定义和使用。

【程序 3-28】

```
PACKAGE pacl IS                                        -- 程序包首开始
  TYPE byte IS RANGE 0 TO 255 ;                        -- 定义数据类型 byte
  SUBTYPE nibble IS byte RANGE 0 TO 15 ;               -- 定义子类型 nibble
CONSTANT byte_ff  : byte := 255 ;                      -- 定义常数 byte_ff
SIGNAL addend : nibble ;                               -- 定义信号 addend
COMPONENT byte_adder                                   -- 定义元件
PORT( a, b : IN byte ;
      c : OUT byte  ;
  overflow : OUT BOOLEAN ) ;
  END COMPONENT ;
  FUNCTION my_function (a : IN byte) Return byte ; -- 定义函数
END pacl ;                                             -- 程序包首结束
```

这显然是一个程序包首，其程序包名是 pacl，在其中定义了一个新的数据类型 byte 和一个子类型 nibble；接着定义了一个数据类型为 byte 的常数 byte_ff 和一个数据类型为 nibble 的信号 addend；还定义了一个元件和函数。由于元件和函数必须有具体的内容，所以将这些内容安排在程序包体中。如果要使用这个程序包中的所有定义，可

利用 USE 语句按如下方式获得访问此程序包的方法。

```
LIBRARY WORK ;
USE WORK.pacl.ALL ;
ENTITY ...
ARCHITHCYURE ...
    ...
```

由于 WORK 库是默认打开的，所以可省去 LIBRARY WORK 语句，只要加入相应的 USE 语句即可。程序 3-29 是另一个在现行 WORK 库中定义程序包并立即使用的示例。

【程序 3-29】

```
PACKAGE  seven IS
   SUBTYPE segments is BIT_VECTOR(0 TO 6) ;
   TYPE bcd IS RANGE 0 TO 9 ;
END seven ;
USE WORK.seven.ALL ;
ENTITY decoder IS
   PORT (input: bcd; drive : out segments) ;
END decoder ;
ARCHITECTURE simple OF decoder IS
BEGIN
   WITH input SELECT
     drive <= B"1111110"  WHEN 0 ,
              B"0110000"  WHEN 1 ,
              B"1101101"  WHEN 2 ,
              B"1111001"  WHEN 3 ,
              B"0110011"  WHEN 4 ,
              B"1011011"  WHEN 5 ,
              B"1011111"  WHEN 6 ,
              B"1110000"  WHEN 7 ,
              B"1111111"  WHEN 8 ,
              B"1111011"  WHEN 9 ,
              B"0000000"  WHEN  OTHERS ;
END simple ;
```

此例是一个可以直接综合的 4 位 BCD 码向 7 段译码显示码转换的 VHDL 描述。此例在程序包 seven 中定义了两个新的数据类型 segments 和 bcd。在 7 段显示译码器 decoder 的实体描述中即使用了这两个数据类型。由于 WORK 库默认是打开的，程序中只加入了 USE 语句。

2. 程序包体

程序包体将包括在程序包首中已定义的子程序的子程序体。程序包体说明部分的组成内容可以是 USE 语句（允许对其它程序包的调用）、子程序定义、子程序体、数据类型说明、子类型说明和常数说明等。对于没有具体子程序说明的程序包体可以省去。

如例 3-28 所示，如果仅仅是定义数据类型或定义数据对象等内容，程序包体是不必要的，程序包首可以独立地被使用；但在程序包中若有子程序说明时，则必须有对应的子程序包体。这时，子程序体必须放在程序包体中。

程序包常用来封装属于多个设计单元分享的信息。

常用的预定义的程序包有：

- STD_LOGIC_1164 程序包

STD_LOGIC_1164 程序包是 IEEE 库中最常用的程序包，是 IEEE 的标准程序包。其中包含了一些数据类型、子类型和函数的定义，这些定义将 VHDL 扩展为一个能描述多值逻辑（即除具有“0”和“1”以外还有其它的逻辑量，如高阻态“Z”、不定态“X”等）的硬件描述语言，很好地满足了实际数字系统的设计需求。STD_LOGIC_1164 程序包中用得最多和最广的是定义了满足工业标准的两个数据类型 STD_LOGIC 和 STD_LOGIC_VECTOR，它们非常适合于 FPGA/CPLD 器件中多值逻辑设计结构。

- STD_LOGIC_ARITH 程序包

STD_LOGIC_ARITH 预先编译在 IEEE 库中，是 Synopsys 公司的程序包。此程序包在 STD_LOGIC_1164 程序包的基础上扩展了三个数据类型 UNSIGNED、SIGNED 和 SMALL_INT，并为其定义了相关的算术运算符和转换函数。

- STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 程序包

STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 程序包都是 Synopsys 公司的程序包，都预先编译在 IEEE 库中。这些程序包重载了可用于 INTEGER 型及 STD_LOGIC 和 STD_LOGIC_VECTOR 型混合运算的运算符，并定义了一个由 STD_LOGIC_VECTOR 型到 INTEGER 型的转换函数。这两个程序包的区别是，STD_LOGIC_SIGNED 中定义的运算符考虑到了符号，是有符号数的运算。

程序包 STD_LOGIC_ARITH、STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 虽然未成为 IEEE 标准，但已经成为事实上的工业标准，绝大多数的 VHDL 综合器和 VHDL 仿真器都支持它们。

- STANDARD 和 TEXTIO 程序包

以上已经提到了 STANDARD 和 TEXTIO 程序包，它们都是 STD 库中的预编译程序包。STANDARD 程序包中定义了许多基本的数据类型、子类型和函数。由于 STANDARD 程序包是 VHDL 标准程序包，实际应用中已隐性地打开了，所以不必再用 USE 语句另作声明。TEXTIO 程序包定义了支持文本文件操作的许多类型和子程序。在使用本程序包之前，需加语句 USE STD.TEXTIO.ALL 。

TEXTIO 程序包主要仅供仿真器使用。可以用文本编辑器建立一个数据文件，文件中包含仿真时需要的数据，然后仿真时用 TEXTIO 程序包中的子程序存取这些数据。在 VHDL 综合器中，此程序包被忽略。

§ 3.8 配置（CONFIGURATION）

配置可以把特定的结构体关联到（指定给）一个确定的实体。正如“配置”一词本身的含义一样，配置语句就是用来为较大的系统设计提供管理和工程组织的。通常在大而复

杂的 VHDL 工程设计中，配置语句可以为实体指定或配属一个结构体，如可以利用配置使仿真器为同一实体配置不同的结构体以使设计者比较不同结构体的仿真差别，或者为例化的各元件实体配置指定的结构体，从而形成一个所希望的例化元件层次构成的设计实体。

配置也是 VHDL 设计实体中的一个基本单元，在综合或仿真中，可以利用配置语句为确定整个设计提供许多有用信息。例如对以元件例化的层次方式构成的 VHDL 设计实体，就可以把配置语句的设置看成是一个元件表，以配置语句指定在顶层设计中的每一元件与一特定结构体相衔接，或赋予特定属性。配置语句还能用于对元件的端口连接进行重新安排等。VHDL 综合器允许将配置规定对一个设计实体中的最高层设计单元，但只支持对最顶层的实体进行配置。但是，通常情况下，配置主要用在 VHDL 的行为仿真中。

配置语句的一般格式如下：

```
CONFIGURATION 配置名 OF 实体名 IS
  配置说明
END 配置名;
```

配置主要为顶层设计实体指定结构体，或为参与例化的元件实体指定所希望的结构体，以层次方式来对元件例化作结构配置。如前所述，每个实体可以拥有多个不同的结构体，而每个结构体的地位是相同的，在这种情况下，可以利用配置说明为这个实体指定一个结构体。程序 3-30 是一个配置的简单方式应用，即在一个描述与非门 nand 的设计实体中会有两个以不同的逻辑描述方式构成的结构体，用配置语句来为特定的结构体需求作配置指定。

【程序 3-30】

```
LIBRARY IEEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY nand IS
   PORT (a : IN STD_LOGIC ;
         b : IN STD_LOGIC ;
         c : OUT STD_LOGIC ) ;
END ENTITY nand ;
ARCHITECTURE one OF nand IS
  BEGIN
    c <= NOT (a AND b) ;
  END ARCHITECTURE one ;
ARCHITECTURE two OF nand IS
   BEGIN
     c <=  '1'  WHEN  (a = '0')AND(b = '0')  ELSE
           '1'  WHEN  (a = '0')AND(b = '1')  ELSE
           '1'  WHEN  (a = '1')AND(b = '0')  ELSE
           '0'  WHEN  (a = '1')AND(b = '1')  ELSE
           '0' ;
   END ARCHITECTURE two ;
CONFIGURATION second OF nand IS
   FOR two
   END FOR ;
  END second ;
```

```
    CONFIGURATION first OF nand IS
      FOR one
      END FOR ;
  END first ;
```

在程序 3-30 中若指定配置名为 second，则为实体 nand 配置的结构体为 two；若指定配置名为 first，则为实体 nand 配置的结构体为 one。这两种结构的描述方式是不同的，但具有相同的逻辑功能。

如果将程序 3-30 中的配置语言全部除去，则可以用此具有两个结构体的实体 nand 构成另一个更高层次设计实体中的元件，并由此设计实体中的配置语句来指定元件实体 nand 使用哪一个结构体。程序 3-31 就是利用程序 3-30 的文件 nand 实现 RS 触发器设计的。最后利用配置语句指定元件实体 nand 中的第二个结构体 two 来构成 nand 的结构体。

【程序 3-31】

```
LIBRARY LEEE;
USE LEEEE.STD_LOGIC_1164.ALL ;
ENTITY rs1 IS
  PORT (  r : IN STD_LOGIC ;
          s : IN STD_LOGIC ;
          q : OUT STD_LOGIC ;
         qf : OUT STD_LOGIC) ;
END rs1 ;
ARCHITECTURE rsf OF rs1 IS
COMPONENT nand
  PORT ( a : IN STD_LOGIC ;
          b : IN STD_LOGIC ;
         c : OUT STD_LOGIC ;
 END COMPONENT ;
BEGIN
  U1:  nand PORT MAP ( a => s, b => qf, c => q ) ;
  U2:  nand PORT MAP ( a => q, b => r, c => qf ) ;
END rsf  ;
CONFIGURATION sel OF rs1 IS
  FOR rsf
    FOR  u1, u2 : nand
    USE ENTITY WORK.nand( two ) ;
    END FOR ;
   END FOR ;
END sel ;
```

这里假设与非门的设计实体已进入工作库 WORK。

【习题】

3-1 简述实体（ENTITY）描述与原理图的关系、结构体描述与原理图的关系。

3-2 子程序调用与元件例化有何区别？函数与过程在具体使用上有何不同？

3-3 VHDL 自上而下系统设计功能的语言结构的基石是什么？

3-4 是否有这样的可能，PROCESS 的运行状态已结束，即已从运行状态进入等待状态，而 PROCESS 中的某条赋值语句尚未完成赋值操作？为什么？从行为仿真和电路实现两方面来谈。

3-5 类属参量与常数有何区别？与原理图输入法相比，类属参量语句的特点为 VHDL 程序设计带来怎样的便利？

3-6 什么是重载函数？重载算符有何用处？如何调用重载算符函数？

3-7 写出 8 位锁存器（如 74LS373）的实体，输入为 D、CLOCK 和 OE，输出为 Q。

3-8 画出与下例实体描述对应的原理图符号：

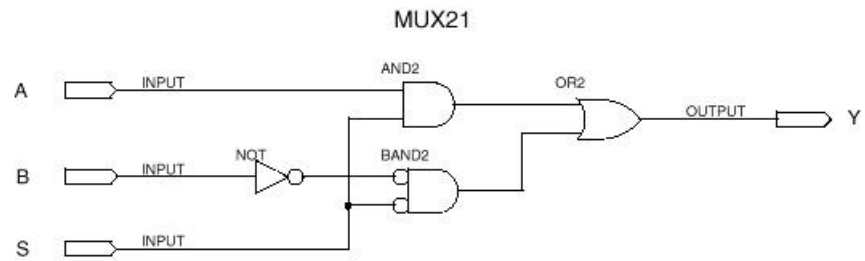
（1）

```
ENTITY buf3s IS                    -- 三态缓冲器
   PORT (input : IN STD_LOGIC ;        -- 输入端
         enable : IN STD_LOGIC ;       -- 使能端
         output : OUT STD_LOGIC ) ;    -- 输出端
END buf3x ;
```

（2）

```
ENTITY mux21 IS                    -- 2 选 1 多路选择器
   PORT (in0,                      -- 数据输入 0
         in1,                      -- 数据输入 1
         sel :  IN STD_LOGIC;        -- 选择信号输入
         output : OUT STD_LOGIC);    -- 输出
END mux21 ;
```

3-9 根据下图写出相应的 VHDL 的结构体（内部信号名由读者自己定义）。



3-10 根据如下的 VHDL 描述画出相应的原理图：

```
ENTITY dlatch IS
   PORT( d, cp : IN STD_LOGIC ;
         q, qn : BUFFER STD_LOGIC ) ;
END dlatch ;
ARCHITECTURE one OF dlatch IS
   SINGAL n1, n2 : STD_LOGIC ;
BEGIN
   n1<= (NOT d) NAND cp ;
   n2<= d NAND cp ;q <= qn NAND n1 ;  qn <= q NAND n2 ;
END one ;
```

3-11 下面是一个简单的 VHDL 描述，请画出其实体（ENTITY）对应的原理图符号，并画出与结构体相应的电路原理图。

```
ENTITY SN74LS20 IS
```

```
    PORT ( I1A, I1B, I1C, I1D : IN STD_LOIGC ;
           I2A, I2B, I2C, I2D : IN STD_LOIGC ;
           O1, O2 : OUT STD_LOGIC  ) ;
END SN74LS20 ;
ARCHITECTURE struc OF SN74LS20 IS
BEGIN
   O1 <= NOT (I1A AND I1B AND I1C AND I1D) ;
   O2 <= NOT (I1A AND I1B AND I1C AND I1D) ;
END struc ;
```

3-12 **在 VHDL 程序中配置有何用处**？

3-13 **嵌套 BLOCK 的可视性规则是什么？以嵌套 BLOCK 的语句方式设计三个并列的 3 输入或门**。

3-14 **叙述函数与过程的异同点、过程与进程的异同点**。